Neural Ordinary Differential Equations: Technical Notes and Implementation Guide

Krishna Kumar

November 5, 2025

Abstract

This document provides comprehensive technical notes on Neural Ordinary Differential Equations (Neural ODEs), covering the theoretical foundations, the adjoint sensitivity method for memory-efficient training, and advanced applications including Latent ODEs and Function Encoders. These notes accompany the Neural ODE lecture slides and provide detailed derivations and implementation insights.

Contents

1 From ResNets to Continuous Dynamics

1.1 The Core Insight

The core idea is a simple but powerful connection between two fields: deep learning and differential equations.

A Residual Network (ResNet) is built from a discrete transformation:

$$h_{t+1} = h_t + f(h_t, \theta_t) \tag{1}$$

You take the current state h_t, add a change f (the "residual"), and you get the next state h_{t+1}. This is a discrete difference equation.

Now, look at the simplest numerical method for solving an Ordinary Differential Equation (ODE), Euler's method:

$$x_{k+1} = x_k + \Delta t \cdot f(x_k, t_k, \theta) \tag{2}$$

These two equations are almost identical. A ResNet is just Euler's method with a step size of $\Delta t = 1$.

The Neural ODE proposal is this: why use a fixed, discrete, and somewhat crude integration scheme? Why not learn the continuous dynamics directly?

We replace the discrete-layer function with a function that defines the derivative:

$$\frac{dh(t)}{dt} = f(h(t), t, \theta) \tag{3}$$

Instead of a 50-layer network, you have a single "ODE block" that computes the continuous transformation of the state $h(t)$ from a start time t_0 to an end time t_1 by solving this differential equation.

1.2 Understanding the ResNet Formula

Let's understand why this question matters. In a ResNet, we're adding a small correction $f(h_t, \theta_t)$ to our hidden state. Think of this like taking steps on a path: we're at position h_t, and we move by some amount $f(h_t, \theta_t)$.

This is exactly the setup for calculus! When we have many small discrete steps, we get a continuous process. The transformation becomes: $h_{t+1} - h_t = f(h_t, \theta_t)$. If we divide by the step size $\Delta t = 1$ and take the limit as steps become infinitesimal, we get $\frac{dh}{dt} = f(h, \theta)$.

Why This Matters: In ResNets, each layer performs the same basic operation (add residual) but with different parameters θ_t. We're essentially discretizing time into layer indices. As the network gets deeper (more layers), it's like taking finer time steps.

The Continuous Limit: If we fix the total "transformation distance" but use infinitely many infinitesimal steps, we get a continuous trajectory. The path h traces out becomes a solution to an ODE.

This isn't just a mathematical curiosity. It means:

- ResNets implicitly define a kind of trajectory through representation space
- The "depth" of the network corresponds to the length of time we integrate
- Different network architectures are different discretizations of continuous dynamics

1.3 The Discrete Nature of ResNets

A standard ResNet layer performs:

$$h_{k+1} = h_k + f(h_k, \theta_k)$$

Key observations:

1. **Fixed depth:** The number of transformations (layers) is predetermined at design time
2. **Fixed step size:** Each layer applies the same magnitude of change (no Δt multiplier)
3. **Different parameters per layer:** Each θ_k is different, meaning the transformation function changes at each step
4. **No adaptive resolution:** You can't adjust the granularity of the transformation based on the difficulty of the problem

Think of it like: you're climbing a mountain, and someone tells you "take exactly 50 steps of exactly 1 meter each, regardless of whether the terrain is steep or flat."

1.4 The Continuous Alternative

Neural ODEs say: instead of taking N fixed discrete steps, model the continuous trajectory:

$$\frac{dh(t)}{dt} = f(h(t), t, \theta)$$

Key differences:

1. **Adaptive depth:** The ODE solver can take as many (or as few) steps as needed for a desired accuracy
2. **Adaptive step size:** Modern solvers use small steps where f changes quickly, large steps where it's smooth
3. **Shared parameters:** The same function f_θ is used throughout. We're not learning N different functions, just one
4. **Continuous representation:** You can query $h(t)$ at *any* time $t \in [t_0, t_1]$, not just at discrete layer indices

The mountain climbing analogy: "walk from point A to point B following this velocity field, adjusting your step size as needed to stay on the path."

1.5 Why "Euler is the Worst"

As Steve Brunton emphasizes: Euler's method is pedagogically important but practically terrible for most problems.

The Problem: Euler's method has first-order error: Error $= \mathcal{O}(\Delta t^2)$ per step. Over N steps covering time $T = N\Delta t$, total error is $\mathcal{O}(\Delta t) = \mathcal{O}(T/N)$.

To get accuracy ϵ, you need $N = \mathcal{O}(T/\epsilon)$ steps. This scales poorly.

Better Methods:

- RK2 (midpoint): Error $\mathcal{O}(\Delta t^3)$ per step, total $\mathcal{O}(\Delta t^2)$
- RK4: Error $\mathcal{O}(\Delta t^5)$ per step, total $\mathcal{O}(\Delta t^4)$
- Adaptive methods (Dormand-Prince): Adjust Δt on the fly, typically 5th-order accurate

The Cost-Benefit: RK4 requires 4 function evaluations per step but can take much larger steps. For the same accuracy:

- Euler: 10,000 steps $\times$ 1 eval/step = 10,000 function evaluations
- RK4: 100 steps $\times$ 4 eval/step = 400 function evaluations

That's a 25x speedup!

ResNets vs Neural ODEs: A 50-layer ResNet is like taking 50 fixed Euler steps. A Neural ODE with an RK4 solver might take only 10-20 *adaptive* steps to achieve better accuracy. This is one reason Neural ODEs can be more parameter-efficient.

1.6 The Compound Interest Analogy

Brunton uses compound interest to illustrate continuous vs discrete updates.

Discrete (like ResNets): You get paid once per year, and the amount is determined by a yearly interest rate.

Continuous (like Neural ODEs): Interest is compounded continuously. The amount changes smoothly according to $\frac{dM}{dt} = rM(t)$.

The discrete formula is $M_{n+1} = M_n + rM_n = (1+r)M_n$. After n years: $M_n = (1+r)^n M_0$.

The continuous formula gives $M(t) = e^{rt} M_0$.

The continuous version is often easier to analyze (it's just an exponential!) and generalizes better (you can query the value at $t = 2.7$ years, not just integer years).

2 The Training Challenge

2.1 The Problem: Memory Cost

This is a beautiful idea for the forward pass. You just use a standard ODE solver (like Runge-Kutta) to find $h(t_1)$ given $h(t_0)$.

The real challenge is training. To update our network parameters θ, we need the gradient of the loss with respect to those parameters: $\frac{dL}{d\theta}$.

The loss L depends on the final state $h(t_1)$. But $h(t_1)$ depends on the entire continuous path of $h(t)$ from t_0 to t_1.

If we use standard backpropagation (automatic differentiation), the chain rule must be applied through all the internal steps of the ODE solver.

An adaptive solver might take $N = 1000$ tiny steps to get an accurate solution.

Standard backpropagation would need to store the activations for all 1000 steps to compute the gradient on the backward pass.

This has a memory cost of $\mathcal{O}(N \cdot \dim(h))$, which is often prohibitively large. We'll run out of memory.

We need a way to get $\frac{dL}{d\theta}$ with $\mathcal{O}(1)$ memory cost. This is where the Adjoint Method comes from.

2.2 Why Standard Backprop Fails

In a standard neural network, backpropagation computes gradients layer by layer using the chain rule. Each layer stores its activations during the forward pass and uses them during the backward pass.

For Neural ODEs:

- The "layers" are the intermediate states $h(t_1), h(t_2), \ldots, h(t_N)$ computed by the ODE solver
- An adaptive solver might use $N = 1000$ or more time points
- Each point requires storing the full state vector $h(t_i)$ (which might be high-dimensional)
- Total memory: $N \times \dim(h)$

For a 100-layer ResNet, you store 100 activation tensors. Doable.

For a Neural ODE with 1000 solver steps, you'd store 1000 activation tensors. Much worse.

And here's the kicker: you don't control N! The adaptive solver chooses it based on the difficulty of the ODE. For stiff equations, N could be enormous.

This makes standard backpropagation through the solver infeasible for Neural ODEs.

3 The Solution: The Adjoint Method

3.1 Optimal Control Perspective

The Adjoint Method comes from optimal control theory. This is a classic constrained optimization problem:

Goal: Minimize $L(h(t_1))$.

Constraint: The path $h(t)$ must obey our ODE: $\frac{dh}{dt} - f(h, t, \theta) = 0$.

The tool for this is the Lagrange multiplier. Because our constraint applies at every moment in time, our multiplier must also be a function of time. We'll call it $a(t)$, the adjoint state.

We build a new function, the Lagrangian $\mathcal{L}$, which combines our goal and our constraint:

$$\mathcal{L} = L(h(t_1)) - \int_{t_0}^{t_1} a(t)^T \left[\frac{dh}{dt} - f(h, t, \theta) \right] dt \tag{4}$$

Here's the key insight:

Since our state $h(t)$ always follows the ODE, the term in the brackets is always zero. The integral is zero.

This means our new function $\mathcal{L}$ is always equal to our original loss L.

Therefore, the gradient we want, $\frac{dL}{d\theta}$, is exactly equal to $\frac{d\mathcal{L}}{d\theta}$.

We haven't changed the answer, but we've introduced a "helper" function $a(t)$ that we get to choose strategically.

3.2 Deriving the Gradient

When we differentiate $\mathcal{L}$ with respect to θ, we get a messy expression. After using calculus (specifically, integration by parts on the $\int a(t)^T \frac{dh}{dt} dt$ term), we're left with terms of two types:

"Bad" Terms: These depend on $\frac{\partial h}{\partial \theta}$, which is the path-dependent gradient we don't want to store.

"Good" Terms: These depend on $\frac{\partial f}{\partial \theta}$, which is the local gradient of our network and is easy to compute.

The Strategy: We choose our adjoint $a(t)$ to make all the "bad" terms vanish.

This strategic choice gives us two equations that define our adjoint state $a(t)$:

An ODE (the "Adjoint Equation"): This is the dynamics $a(t)$ must follow.

$$\frac{da(t)}{dt} = -a(t)^T \frac{\partial f}{\partial h} \tag{5}$$

A "Starting" Condition: This is the value of $a(t)$ at the end of the forward pass.

$$a(t_1)^T = \frac{\partial L}{\partial h(t_1)} \tag{6}$$

Notice that the "initial" condition for $a(t)$ is given at t_1. This means we must solve this new ODE backward in time, from t_1 to t_0.

3.3 The Final Gradient

After we choose $a(t)$ to make all the "bad" terms disappear, the only term left in our equation for $\frac{dL}{d\theta}$ is the "good" one. This gives us our final, computable gradient:

$$\frac{dL}{d\theta} = \int_{t_0}^{t_1} a(t)^T \frac{\partial f}{\partial \theta} dt \tag{7}$$

This math isn't just a proof; it's a concrete algorithm.

4 The Adjoint Algorithm

4.1 The Concrete Process

This is how you train a Neural ODE with $\mathcal{O}(1)$ memory.

Forward Pass:

1. Start with your input $h(t_0)$.
2. Use a standard ODE solver to integrate $\frac{dh}{dt} = f(h, t, \theta)$ forward in time from t_0 to t_1.
3. Get the final state $h(t_1)$ and compute your loss $L(h(t_1))$.
4. **Crucially:** Save only the final state $h(t_1)$ and the inputs $h(t_0), t_0, t_1$. Discard all intermediate steps.

Backward Pass (The Adjoint Solve):

1. **Get Initial Adjoint State.** Compute $a(t_1) = \frac{\partial L}{\partial h(t_1)}$. This is just the standard gradient of your loss function.

2. **Solve Augmented ODE Backward.** We now solve a new, augmented ODE system backward in time from t_1 to t_0. This system has three components that we solve simultaneously:

$$\text{Adjoint State:} \quad \frac{da}{dt} = -a(t)^T \frac{\partial f}{\partial h} \tag{8}$$

$$\text{Parameter Gradients:} \quad \frac{d(\frac{dL}{d\theta})}{dt} = -a(t)^T \frac{\partial f}{\partial \theta} \tag{9}$$

$$\text{Original State:} \quad \frac{dh}{dt} = -f(h, t, \theta) \tag{10}$$

Wait, why do we solve for h again? Because we didn't save the path. To compute $\frac{\partial f}{\partial h}$ and $\frac{\partial f}{\partial \theta}$ at time t during the backward pass, we need the value of $h(t)$. We recompute it on-the-fly by solving its ODE backward as well.

Done:

When your backward solver reaches t_0, the variable $\frac{dL}{d\theta}$ will have accumulated the full, correct gradient.

You can now use this gradient to take a step with your optimizer (e.g., Adam, SGD).

We have successfully traded a massive $\mathcal{O}(N)$ memory cost for an $\mathcal{O}(1)$ memory cost. The "price" was the extra computation of solving the second, more complex ODE system in reverse.

4.2 Understanding the Augmented System

The backward pass solves an augmented ODE with four state variables:

$$\frac{d}{dt} \begin{bmatrix} h(t) \\ a(t) \\ \frac{\partial L}{\partial \theta} \\ \frac{\partial L}{\partial t_0} \end{bmatrix} = \begin{bmatrix} -f(h, t, \theta) \\ -a(t)^T \frac{\partial f}{\partial h} \\ -a(t)^T \frac{\partial f}{\partial \theta} \\ -a(t)^T f \end{bmatrix} \tag{11}$$

Why this works: The adjoint method comes from optimal control theory. We're computing gradients of an integral functional.

Forward Pass (Step 1-2): Standard. Run the ODE solver to get the final state and compute the loss. Nothing special here.

Backward Pass Setup: We want three gradients: (1) $\frac{\partial L}{\partial \theta}$ to update parameters, (2) $\frac{\partial L}{\partial t_0}$ if initial time is learned, (3) adjoint $a(t) = \frac{\partial L}{\partial h(t)}$ to propagate gradients through time.

The Augmented State: We pack all three quantities into a single vector $s(t)$ and solve one big ODE for all of them simultaneously. This is computationally efficient.

Initialization: At $t = t_1$ (end of forward pass), we know $\frac{\partial L}{\partial h(t_1)}$ from automatic differentiation. The gradient w.r.t. parameters and initial time start at zero and accumulate as we integrate backward.

4.3 The Dynamics of Each Component

Adjoint evolution: $\frac{da}{dt} = -a^T \frac{\partial f}{\partial h}$. This propagates the gradient backward through the dynamics. The Jacobian $\frac{\partial f}{\partial h}$ tells us how changes in h affect the derivative f. The negative sign comes from integrating backward in time.

Parameter gradients: $\frac{d}{dt} \frac{\partial L}{\partial \theta} = -a^T \frac{\partial f}{\partial \theta}$. At each time t, we accumulate how the loss depends on parameters. The term $\frac{\partial f}{\partial \theta}$ is how parameters affect the dynamics, weighted by the adjoint $a(t)$.

Initial time gradient: $\frac{d}{dt}\frac{\partial L}{\partial t_0} = -a^T f$. This accounts for how changing the start time affects the loss.

Key Insight: All these derivatives are computed using *vector-Jacobian products* (VJPs): $a^T \frac{\partial f}{\partial h}$. PyTorch autograd computes VJPs efficiently via reverse-mode AD without ever forming the full Jacobian matrix. This is why the method is practical for high-dimensional systems.

Why Autodiff is Crucial: Since f is a neural network, we get $\frac{\partial f}{\partial h}$ and $\frac{\partial f}{\partial \theta}$ for FREE from PyTorch's autograd! We don't need to compute these by hand (which would be impossible). This is what makes the adjoint method practical – we use the autodifferentiability of the neural network to get the equations for the Lagrange multiplier.

Final Output: After integrating from t_1 to t_0, we have $\frac{\partial L}{\partial \theta}(t_0)$ – the gradient we need for the optimizer!

5 Detailed Adjoint Derivation via Lagrange Multipliers

5.1 The Flow Map

Understanding the Flow Map: The flow map Φ is a central concept from dynamical systems theory. It maps initial conditions forward in time according to the differential equation.

Think of it like this: if you drop a leaf in a river at position $z(t_0)$ at time t_0, the flow map tells you where that leaf will be at time t_1. The river's current is the vector field $f(z, t, \theta)$.

Why "Flow"? Because the ODE defines a flow in state space. Particles flow along trajectories determined by $\frac{dz}{dt} = f(z, t, \theta)$.

The Integral Form: The equation $z(t_1) = z(t_0) + \int_{t_0}^{t_1} f(z(\tau), \tau, \theta)d\tau$ is the fundamental theorem of calculus applied to ODEs:

$$z(t_1) - z(t_0) = \int_{t_0}^{t_1} \frac{dz}{d\tau}d\tau = \int_{t_0}^{t_1} f(z(\tau), \tau, \theta)d\tau$$

Dependence on Parameters: The flow map depends on θ because the vector field f depends on θ. Changing θ changes the "currents" and thus where particles end up.

The Challenge: To train our neural network (optimize θ), we need $\frac{dL}{d\theta}$. But L depends on θ through this complex integral! We need to differentiate through the ODE solver.

Naive Approach (Bad): Backpropagate through every step of the ODE solver $\rightarrow$ O(N) memory where N = number of steps. For adaptive solvers, N could be 100s or 1000s!

Smart Approach (Good): Use optimal control theory and Lagrange multipliers $\rightarrow$ O(1) memory!

5.2 The Gradient Challenge

What we want: $\frac{dL}{d\theta}$ where $L = L(\Phi(z(t_0), t_0, t_1, \theta))$

Chain rule attempt:

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial z(t_1)}\frac{\partial z(t_1)}{\partial \theta} \tag{12}$$

The problem: What is $\frac{\partial z(t_1)}{\partial \theta}$?

By the chain rule through the integral:

$$\frac{\partial z(t_1)}{\partial \theta} = \int_{t_0}^{t_1} \frac{\partial f}{\partial \theta}dt + \int_{t_0}^{t_1} \frac{\partial f}{\partial z}\frac{\partial z(\tau)}{\partial \theta}d\tau$$

This is circular! To find $\frac{\partial z(t_1)}{\partial \theta}$, we need $\frac{\partial z(\tau)}{\partial \theta}$ for all $\tau \in [t_0, t_1]$.

Why this is bad:

- We'd need to solve another ODE (called the "sensitivity equation") forward in time
- This ODE has dimension $\dim(z) \times \dim(\theta)$, which can be huge
- We'd need to store the entire path $z(\tau)$ to compute $\frac{\partial f}{\partial z}$ at each point

Key Insight: Instead of pushing derivatives forward (which requires storing the path), we'll pull them backward (which only requires the endpoint).

5.3 Lagrange Multipliers: The Setup

Key Idea: Turn constraint into penalty using Lagrange multiplier $\lambda(t)$

The ODE $\frac{dz}{dt} = f(z, t, \theta)$ is a constraint that must hold at every time t.

We introduce a Lagrange multiplier function $\lambda(t)$ (one for each time point) and form:

$$\mathcal{L} = L(z(t_1)) - \int_{t_0}^{t_1} \lambda(t)^T \left[\frac{dz}{dt} - f(z(t), t, \theta) \right] dt \tag{13}$$

Why this helps: Since the constraint is always satisfied, the integral is always zero. So $\mathcal{L} = L$. But when we take derivatives, we get extra terms involving λ that we can use strategically.

The Plan: Take $\frac{d\mathcal{L}}{d\theta}$, move derivatives around using integration by parts, then choose λ to eliminate the hard terms.

5.4 Deriving the Adjoint Equation (Step 1: Integration by Parts)

Starting point:

$$\frac{d\mathcal{L}}{d\theta} = \frac{\partial L}{\partial z(t_1)} \frac{\partial z(t_1)}{\partial \theta} - \int_{t_0}^{t_1} \lambda(t)^T \left[\frac{\partial}{\partial \theta} \frac{dz}{dt} - \frac{\partial f}{\partial \theta} - \frac{\partial f}{\partial z} \frac{\partial z}{\partial \theta} \right] dt$$

The problematic term is $\lambda^T \frac{\partial}{\partial \theta} \frac{dz}{dt} = \lambda^T \frac{d}{dt} \frac{\partial z}{\partial \theta}$ (we can swap $\frac{\partial}{\partial \theta}$ and $\frac{d}{dt}$).

Integration by parts: Use $\int u \frac{dv}{dt} = uv\big|_{t_0}^{t_1} - \int v \frac{du}{dt}$ with $u = \lambda(t)^T$ and $v = \frac{\partial z}{\partial \theta}$:

$$\int_{t_0}^{t_1} \lambda(t)^T \frac{d}{dt} \frac{\partial z}{\partial \theta} dt = \lambda(t_1)^T \frac{\partial z(t_1)}{\partial \theta} - \lambda(t_0)^T \frac{\partial z(t_0)}{\partial \theta} - \int_{t_0}^{t_1} \frac{d\lambda}{dt}^T \frac{\partial z}{\partial \theta} dt \tag{14}$$

Why this is useful: We've moved the derivative from z to λ. Now we'll choose λ strategically.

5.5 Deriving the Adjoint Equation (Step 2: Grouping Terms)

After integration by parts and rearranging, we get:

$$\frac{d\mathcal{L}}{d\theta} = \left[\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)^T \right] \frac{\partial z(t_1)}{\partial \theta} \tag{15}$$

$$+ \lambda(t_0)^T \frac{\partial z(t_0)}{\partial \theta} \tag{16}$$

$$+ \int_{t_0}^{t_1} \left[\frac{d\lambda}{dt}^T + \lambda(t)^T \frac{\partial f}{\partial z} \right] \frac{\partial z}{\partial \theta} dt \tag{17}$$

$$+ \int_{t_0}^{t_1} \lambda(t)^T \frac{\partial f}{\partial \theta} dt \tag{18}$$

The terms:

1. Boundary term at t_1: involves $\frac{\partial z(t_1)}{\partial \theta}$ (hard!)
2. Boundary term at t_0: involves $\frac{\partial z(t_0)}{\partial \theta} = 0$ (initial condition doesn't depend on θ)
3. Integral term: involves $\frac{\partial z}{\partial \theta}$ throughout time (hard!)
4. Final term: involves only $\frac{\partial f}{\partial \theta}$ (easy! This is local gradient)

Strategy: Choose λ to make terms 1 and 3 vanish.

5.6 The Adjoint Solution: Making Terms Vanish

Boundary condition at t_1: Choose

$$\lambda(t_1)^T = \frac{\partial L}{\partial z(t_1)} \tag{19}$$

This makes the first term zero.
Dynamics of $\lambda(t)$ (adjoint equation): Choose

$$\frac{d\lambda}{dt}^T = -\lambda(t)^T \frac{\partial f}{\partial z} \tag{20}$$

This makes the integral term (term 3) zero.
Final Gradient: With these choices, the only surviving term is:

$$\frac{dL}{d\theta} = \int_{t_0}^{t_1} \lambda(t)^T \frac{\partial f}{\partial \theta} dt \tag{21}$$

The algorithm:

1. Forward: Solve $\frac{dz}{dt} = f(z, t, \theta)$ from t_0 to t_1
2. Compute $\lambda(t_1) = \left(\frac{\partial L}{\partial z(t_1)}\right)^T$
3. Backward: Solve $\frac{d\lambda}{dt} = -\left(\frac{\partial f}{\partial z}\right)^T \lambda$ from t_1 to t_0
4. Accumulate $\frac{dL}{d\theta} = \int_{t_0}^{t_1} \lambda^T \frac{\partial f}{\partial \theta} dt$

5.7 How Neural ODEs Automate This

Why Autodiff Makes This Practical:
In the mathematical derivation, we need:

- $\frac{\partial f}{\partial z}$: the Jacobian of the network w.r.t. its input
- $\frac{\partial f}{\partial \theta}$: the Jacobian of the network w.r.t. its parameters

For a neural network, these would be nightmarish to compute by hand!
Enter PyTorch Autograd:
Modern autodiff libraries (PyTorch, JAX) provide automatic computation of:

- **Vector-Jacobian Products (VJPs):** Given a vector v and function f, compute $v^T \frac{\partial f}{\partial x}$ without forming the full Jacobian
- This is exactly what reverse-mode autodiff does!

The connection:

- $\lambda(t)^T \frac{\partial f}{\partial z}$ is a VJP with vector λ
- $\lambda(t)^T \frac{\partial f}{\partial \theta}$ is a VJP with vector λ
- PyTorch computes both efficiently via `backward()`

The practical algorithm:

1. Wrap your ODE solver in PyTorch
2. During backward pass, PyTorch automatically:
   - Computes $\frac{\partial L}{\partial z(t_1)}$ (standard backprop)
   - Calls the adjoint solver with this as $\lambda(t_1)$
   - Uses VJPs to compute $\frac{d\lambda}{dt}$ at each time step
   - Accumulates gradient $\frac{dL}{d\theta}$
3. You get the correct gradient without any manual Jacobian calculations!

Memory efficiency: Only need to store:

- Current state $z(t)$ (recomputed backward)
- Current adjoint $\lambda(t)$
- Accumulated gradient $\frac{dL}{d\theta}$

Total: $\mathcal{O}(\dim(z) + \dim(\theta))$ memory, independent of number of solver steps!

6 Latent ODEs for Irregular Time Series

6.1 The Motivation

Standard RNNs assume observations come at regular, fixed intervals. Real-world time series data is often:

- Irregularly sampled (observations at random times)
- Missing data (gaps in the sequence)
- Variable sampling rates (dense in some periods, sparse in others)

Examples: medical records (vitals measured when patient visits), sensor data (logged when event detected), financial data (transactions occur irregularly).

The Problem with RNNs: Standard RNNs have no explicit notion of time. They process a sequence $x_1, x_2, \ldots, x_N$ but don't know that x_5 and x_6 might be 10 days apart while x_2 and x_3 are 1 hour apart.

Attempts to fix this (time-aware RNNs, etc.) are often ad-hoc and don't generalize well.

6.2 The Latent ODE Solution

Core Idea: Model the underlying continuous dynamics in a latent space. Observations are noisy snapshots of this continuous trajectory.

Three-Component Architecture:

1. **ODE-RNN Encoder:** Process observations backward in time, using ODEs to handle irregular gaps
2. **Latent ODE:** Continuous dynamics in latent space: $\frac{dz}{dt} = f_\theta(z, t)$
3. **Decoder:** Map latent states to observations: $p(x_i|z(t_i))$

6.3 The ODE-RNN Encoder

Why Process Backward? This is a key design choice! Processing observations backward in time naturally produces an initial condition z_0 that encodes the entire sequence. Think of it like reverse-engineering the initial state from the trajectory.

The ODE-RNN Step: At each observation time t_i (going backward):

1. Evolve hidden state from t_{i+1} to t_i using an ODE: this accounts for the time gap
2. Update with RNN cell using observation x_i: this incorporates the data

This is different from standard RNNs which assume fixed time steps. The ODE naturally handles variable gaps!

Why This Architecture Works:

- Encoder handles irregularity by explicitly modeling time via ODEs
- Latent space has smooth, continuous dynamics (good inductive bias)
- Can query $z(t)$ at any time, not just observation points
- Decoder can make predictions at arbitrary future times

6.4 VAE Framework for Latent ODEs

VAE Framework: This is trained as a variational autoencoder:

Encoder: $q_\phi(z_0|x_{1:N})$ produces a distribution over initial states (typically Gaussian with learned mean and variance)

Prior: $p(z_0) = \mathcal{N}(0, I)$ (standard normal)

Generative Model:

1. Sample $z_0 \sim p(z_0)$
2. Evolve via ODE to get $z(t_i)$ for each observation time
3. Generate observations $x_i \sim p(x|z(t_i))$

Training Objective (ELBO):

$$\mathcal{L} = \mathbb{E}_{q(z_0)}[\sum_{i=1}^{N} \log p(x_i|z(t_i))] - \mathrm{KL}(q(z_0)\|p(z_0))$$

The first term is reconstruction: how well can we predict observations from latent states? The second term is regularization: keep the latent distribution close to the prior.

Key Advantage: The KL term is computed only on z_0, not on the entire trajectory. The ODE deterministically maps z_0 to all future $z(t)$, so we only need uncertainty about the initial state!

7 The ELBO Loss Function

7.1 What is ELBO?

The Evidence Lower Bound is the objective function for variational autoencoders (VAEs). We can't directly optimize the data likelihood $p(x)$ because it requires integrating over all possible latent states. Instead, we introduce an approximate posterior $q(z_0|x)$ and maximize a lower bound.

Derivation Sketch:

$$\begin{aligned}\log p(x) &= \log \int p(x|z_0)p(z_0)dz_0 \\ &\geq \mathbb{E}_{q(z_0|x)}[\log p(x|z_0)] - D_{KL}(q(z_0|x)\|p(z_0))\end{aligned}$$

The inequality comes from Jensen's inequality. Maximizing the ELBO is equivalent to minimizing the gap between $q(z_0|x)$ and the true posterior $p(z_0|x)$.

7.2 Why Two Terms?

Reconstruction Term: $\mathbb{E}_{q(z_0|x)}[\sum_{i=1}^{N} \log p(x_i|z(t_i))]$

This says: "Sample z_0 from the encoder, evolve it via the ODE to get $z(t_i)$ at each observation time, then decode to get predictions. How likely are the actual observations under these predictions?"

In practice:

1. Encoder outputs μ_{z_0}, σ_{z_0}
2. Sample $z_0 \sim \mathcal{N}(\mu_{z_0}, \sigma_{z_0}^2)$ (reparameterization trick)
3. Solve ODE: $z(t_i) = z_0 + \int_0^{t_i} f_\theta(z(\tau), \tau)d\tau$
4. Compute $\hat{x}_i = \text{Decoder}(z(t_i))$
5. Loss: $-\sum_i \frac{\|x_i - \hat{x}_i\|^2}{2\sigma_{\text{noise}}^2}$

KL Regularization: $D_{KL}(q(z_0|x)\|p(z_0))$

This says: "Don't let the encoder produce arbitrary distributions. Keep it close to standard normal."

Why? Without this term, the encoder could make $\sigma_{z_0} \to 0$ (no uncertainty) and memorize the data. The KL term forces the latent space to be smooth and structured.

For Gaussian distributions, this has a closed form:

$$D_{KL}(\mathcal{N}(\mu, \sigma^2)\|\mathcal{N}(0,1)) = \frac{1}{2}\sum_{j=1}^{d}(\mu_j^2 + \sigma_j^2 - \log\sigma_j^2 - 1)$$

7.3 Key Insight for Latent ODEs

The KL term is only on z_0, not on the entire trajectory $\{z(t_i)\}$!

In a standard VAE with discrete observations, you'd need to regularize the distribution at each time step. But in Latent ODEs, the dynamics are deterministic:

$$z(t_i) = \Phi(z_0, 0, t_i, \theta)$$

where Φ is the ODE flow map. So all the randomness comes from z_0. This is computationally efficient and also a good inductive bias: the world evolves deterministically, uncertainty comes from not knowing the initial state.

7.4 Training Algorithm

1. Forward pass:
   - Run ODE-RNN encoder backward to get $q(z_0|x) = \mathcal{N}(\mu_{z_0}, \sigma^2_{z_0})$
   - Sample z_0 using reparameterization trick
   - Solve ODE forward to get $\{z(t_i)\}$
   - Decode to get predictions $\{\hat{x}_i\}$
2. Compute ELBO:
$$\mathcal{L} = \sum_i \log p(x_i|z(t_i)) - D_{KL}(q(z_0|x)\|\mathcal{N}(0, I))$$
3. Backward pass: Use adjoint method to compute $\frac{\partial \mathcal{L}}{\partial \theta}$
4. Update parameters with gradient ascent (maximize ELBO)

Comparison to Standard RNN: A standard RNN would have a latent state at each time step with no constraints between them. Latent ODE enforces that the latent trajectory follows smooth dynamics, which is a strong inductive bias for physical systems and time series with underlying continuous processes.

8 Function Encoders for Zero-Shot Transfer

8.1 The Zero-Shot Transfer Problem

Scenario: You've trained Neural ODE models on multiple dynamical systems. Now you see a NEW system (never seen during training). Can you make predictions without retraining?

Traditional Approach: Train a new model from scratch on the new system. Expensive and slow.

Function Encoder Approach: Learn a set of "basis" Neural ODEs during training. For a new system, find the right combination of these basis functions. No training needed!

Training Phase:

- Train on multiple systems: $\{S_1, S_2, \ldots, S_K\}$
- Each system has multiple demonstration trajectories
- Learn basis functions $\phi_1, \ldots, \phi_K$ that can represent all training systems as linear combinations

Test Phase (Zero-Shot):

- Given a NEW system (never seen during training)
- Observe a few demonstration trajectories
- Compute coefficients α_i to represent the new system
- Make predictions using $\sum \alpha_i \phi_i$

The key is: computing coefficients is fast (no training), but they can represent complex dynamics!

8.2 Hilbert Space Formulation

Hilbert Space Setup: We need to define an inner product between dynamical systems. For functions, $\langle f, g \rangle = \int f(x)g(x)dx$. For dynamics?

A dynamical system is characterized by its velocity field $v(z,t) = \frac{dz}{dt}$. We define the inner product as:

$$\langle v_1, v_2 \rangle = \int_0^T \int_{\mathcal{Z}} v_1(z,t) \cdot v_2(z,t)\, p(z,t)\, dz\, dt$$

where $p(z,t)$ is a measure over states and time.

For Neural ODE Basis: The basis function ϕ_i has velocity field $f_i(z,t)$. To compute the coefficient for a demonstration trajectory x_{demo}:

$$\alpha_i = \langle \phi_i, x_{\text{demo}} \rangle = \mathbb{E}_{t,z\sim\phi_i}[v_{\text{demo}}(z,t) \cdot f_i(z,t)]$$

where $v_{\text{demo}}(z,t)$ is the velocity of the demonstration at state z and time t.

Monte Carlo Computation:

1. Run the basis ODE ϕ_i to generate trajectory (t_j, z_j) samples
2. At each sample point, evaluate the demonstration's velocity: $v_{\text{demo}}(z_j, t_j)$
3. Compute dot product with basis velocity: $v_{\text{demo}}(z_j, t_j) \cdot f_i(z_j, t_j)$
4. Average over all samples: $\alpha_i \approx \frac{1}{M} \sum_{j=1}^{M} v_{\text{demo}}(z_j, t_j) \cdot f_i(z_j, t_j)$

This is efficient because we only need forward passes through f_i, no backpropagation!

9 Summary

Neural ODEs provide a powerful framework for modeling continuous-time dynamics with deep learning. The key insights are:

1. **Continuous Depth:** Replace discrete layers with continuous dynamics
2. **Memory Efficiency:** Adjoint method enables $\mathcal{O}(1)$ memory training
3. **Adaptive Computation:** ODE solver automatically adjusts resolution
4. **Irregular Time Series:** Latent ODEs handle missing and irregular data naturally

5. **Transfer Learning:** Function encoders enable zero-shot transfer

The adjoint method is the crucial innovation that makes Neural ODEs practical. By reformulating gradient computation as a constrained optimization problem with Lagrange multipliers, we avoid storing the entire forward trajectory and achieve constant memory cost regardless of the number of solver steps.